Heterogeneous programming for hybrid CPU-GPU systems: Lessons learned from computational chemistry

Jeff Hammond and Eugene DePrince

Argonne National Laboratory (LCF and CNM)

(Eugene moved to Georgia Tech last week)

NCAR – 8 September 2011

Abstract (for posterity)

We have implemented complex computational chemistry algorithms known as coupled-cluster theory (a quantum many-body method) using CUDA and OpenMP for efficient execution on hybrid CPU-GPU systems. While many of the floating-point operations of this code are performed inside of the CPU or GPU BLAS library, the large memory footprint of the data-structures requires careful consideration of data motion. The latest version of our code is able to exploit multiple CPU cores and one GPU at the same time, maximizing overlap of communication and computation using CUDA streams as well as dealing with load-balancing via dynamically rescheduling computation between iterations. The techniques used for our computational chemistry application should be applicable to other domains, especially those which have large memory footprints and use iterative solvers. In addition to our computational chemistry application performance results, we will show microbenchmarks relevant to GPU codes executing over multiple nodes using MPI. The role of GPUdirect and related developments in the CUDA software stack will be discussed.

Power

Power efficiency forces us to use “parallelism all the way down”. . .

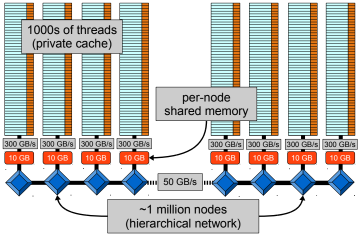
Green500 summary:

1	Blue Gene/Q (2097.19 MF/W)
3	Intel+ATI (1375.88 MF/W)
4	Intel+NVIDIA (958.35 MF/W)
14	POWER7 (565.97 MF/W)
19	Cell-based (458.33 MF/W)
21	Intel-only (441.18 MF/W)

Central Points

Given:

- Power compels "kitchen sink" architecture design with many-level parallelism (evolving from multi-level parallelism).
- CPU+GPU just scratches the surface of programmer pain to come.
- Compilers will never be good enough.
- Memory (bandwidth, capacity, granularity) will *always* be the bottleneck.



Central Points

Try:

- High-level code generation (in addition to low-level code generation).
- Factorize code around inter/intra-node parallelism.
- Dynamic task parallelism on top of data-parallelism
- Asynchrony, overlap, nonblocking, one-sided, etc.
- Performance portability impossible with traditional languages (Fortran, C, C++).

Regarding PGAS, compilers that cannot block for cache should not be allowed to generate network communication.

Tensor Contraction Engine

Tensor Contraction Engine

What does it do?

1. GUI input quantum many-body theory e.g. CCSD.
2. Operator specification of theory (as in a theory paper).
3. Apply Wick's theory to transform operator expressions into array expressions (as in a computational paper).
4. Transform input array expression to operation tree using many types of optimization (i.e. compile).
5. Produce Fortran+Global Arrays+NXTVAL implementation

Developer can intercept at various stages to modify theory, algorithm or implementation.

TCE Input

We get 73 lines of serial F90 or 604 lines of parallel F77 from this:

```
1.0/1.0 Sum( g1 g2 p3 h4 ) f( g1 g2 ) t( p3 h4 ) { g1+ g2
} { p3+ h4 }
1.0/4.0 Sum( g1 g2 g3 g4 p5 h6 ) v( g1 g2 g3 g4 ) t( p5 h6
) { g1+ g2+ g4 g3 } { p5+ h6 }
1.0/16.0 Sum( g1 g2 g3 g4 p5 p6 h7 h8 ) v( g1 g2 g3 g4 )
t( p5 p6 h7 h8 ) { g1+ g2+ g4 g3 } { p5+ p6+ h8 h7 }
1.0/8.0 Sum( g1 g2 g3 g4 p5 h6 p7 h8 ) v( g1 g2 g3 g4 ) t(
p5 h6 ) t( p7 h8 ) { g1+ g2+ g4 g3 } { p5+ h6 } { p7+ h8 }
```

LaTeX equivalent of the first term:

$$\sum_{g_1,g_2,p_3,h_4} f_{g_1,g_2} t_{p_3,h_4} \{g_1^\dagger g_2\}\{p_3^\dagger h_4\}$$

Summary of TCE module

```
http://cloc.sourceforge.net v 1.53  T=30.0 s
-----------------------------------------
Language       files  blank  comment     code
-----------------------------------------
Fortran 77     11451   1004   115129  2824724
-----------------------------------------
SUM:           11451   1004   115129  2824724
-----------------------------------------
```

Only <25 KLOC are hand-written; ∼100 KLOC is utility code following TCE data-parallel template.

Expansion from TCE input to massively-parallel F77 is ∼ 200.

TCE Template

Pseudocode for $R^{a,b}_{i,j} = R^{c,d}_{i,j} * V^{c,d}_{a,b}$:

```
for i,j in occupied blocks:
   for a,b in virtual blocks:
      for c,d in virtual blocks:
         if symmetry_criteria(i,j,a,b,c,d):
            if dynamic_load_balancer(me):
               Get block t(i,j,c,d) from T
               Permute t(i,j,c,d)
               Get block v(a,b,c,d) from V
               Permute v(a,b,c,d)
               r(i,j,c,d) += t(i,j,c,d) * v(a,b,c,d)
      Permute r(i,j,a,b)
      Accumulate r(i,j,a,b) block to R
```

TCE for Heterogeneous Systems

- New `Get` and `Accumulate` that communicate from/to remote memory to GPU instead of CPU.
- Implement `Permute` in GPU code, use GPU BLAS.

In this scenario, porting to a new heterogeneous architecture requires a few hundred lines of code.

This is only a first-order solution:

- Load-balancing is significantly harder when the compute portion goes significantly faster.
- Single-level blocking not optimal for heterogenous nodes.

Quantum Chemistry on Heterogeneous Nodes

Coupled-cluster theory

$$
\begin{aligned}
|CC\rangle &= \exp(T)|0\rangle \\
T &= T_1 + T_2 + \cdots + T_n \quad (n \ll N) \\
T_1 &= \sum_{ia} t_i^a \hat{a}_a^\dagger \hat{a}_i \\
T_2 &= \sum_{ijab} t_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i \\
|\Psi_{CCD}\rangle &= \exp(T_2)|\Psi_{HF}\rangle \\
&= (1 + T_2 + T_2^2)|\Psi_{HF}\rangle \\
|\Psi_{CCSD}\rangle &= \exp(T_1 + T_2)|\Psi_{HF}\rangle \\
&= (1 + T_1 + \cdots + T_1^4 + T_2 + T_2^2 + T_1 T_2 + T_1^2 T_2)|\Psi_{HF}\rangle
\end{aligned}
$$

Coupled cluster (CCD) implementation

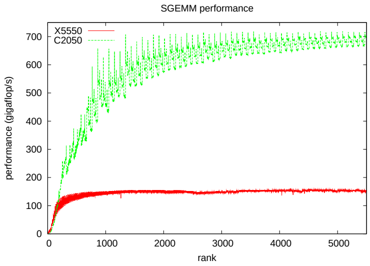
$$R^{ab}_{ij} = V^{ab}_{ij} + P(ia,jb)\Bigg[T^{ae}_{ij}I^{b}_{e} - T^{ab}_{im}I^{m}_{j} + \frac{1}{2}V^{ab}_{ef}T^{ef}_{ij} + \frac{1}{2}T^{ab}_{mn}I^{mn}_{ij} - T^{ae}_{mj}I^{mb}_{ie} - I^{ma}_{ie}T^{eb}_{mj} + (2T^{ea}_{mi} - T^{ea}_{im})I^{mb}_{ej}\Bigg]$$

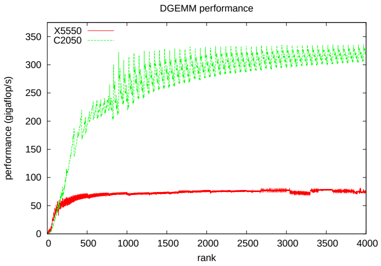
$$\begin{aligned}
I^{a}_{b} &= (-2V^{mn}_{eb} + V^{mn}_{be})T^{ea}_{mn} \\
I^{i}_{j} &= (2V^{mi}_{ef} - V^{im}_{ef})T^{ef}_{mj} \\
I^{ij}_{kl} &= V^{ij}_{kl} + V^{ij}_{ef}T^{ef}_{kl} \\
I^{ia}_{jb} &= V^{ia}_{jb} - \frac{1}{2}V^{im}_{eb}T^{ea}_{jm} \\
I^{ia}_{bj} &= V^{ia}_{bj} + V^{im}_{be}(T^{ea}_{mj} - \frac{1}{2}T^{ae}_{mj}) - \frac{1}{2}V^{mi}_{be}T^{ae}_{mj}
\end{aligned}$$

Tensor contractions currently implemented as `GEMM` plus `PERMUTE`.

Relative Performance of GEMM

GPU versus SMP CPU (8 threads):





CPU = 156.2 GF	CPU = 79.2 GF
GPU = 717.6 GF	GPU = 335.6 GF

We expect roughly 4-5 times speedup based upon this evaluation because `GEMM` *should* be 90% of the execution time.

CPU or GPU CCD

This code was written to minimize communication. Only one term is memory-bound and requires communication during the iteration.

	C2050	C1060	X5550
C_8H_{10}	0.3	0.8	1.3
$C_{10}H_{12}$	0.8	2.5	3.5
$C_{12}H_{14}$	2.0	7.1	10.0
$C_{14}H_{10}$	2.7	10.2	13.9
$C_{14}H_{16}$	4.5	16.7	21.6
$C_{16}H_{18}$	10.5	35.9	50.2
$C_{18}H_{20}$	20.1	73.0	86.6

Iteration time in seconds for double precision.

Numerical Precision versus Performance

Iteration time in seconds

	C1060		C2050		X5550	
molecule	SP	DP	SP	DP	SP	DP
C_8H_{10}	0.2	0.8	0.2	0.3	0.7	1.3
$C_{10}H_{12}$	0.7	2.5	0.4	0.8	2.0	3.5
$C_{12}H_{14}$	1.8	7.1	1.0	2.0	5.6	10.0
$C_{14}H_{10}$	2.6	10.2	1.5	2.7	8.4	13.9
$C_{14}H_{16}$	4.1	16.7	2.4	4.5	12.1	21.6
$C_{16}H_{18}$	9.0	35.9	5.0	10.5	28.8	50.2
$C_{18}H_{20}$	17.2	73.0	10.1	20.1	47.0	86.6

Mixed-precision is essentially trivial when you have a relaxation solver and is always worth at least 2x (except on BG).

CPU and GPU CCSD

This code was rewritten to minimize memory, overlap communication. Focus on overlap and pipelining.

	Hybrid	CPU	Molpro
C_8H_{10}	0.6	1.4	2.4
$C_{10}H_{12}$	1.4	4.1	7.2
$C_{12}H_{14}$	3.3	11.1	19.0
$C_{14}H_{10}$	4.4	15.5	31.0
$C_{14}H_{16}$	6.3	24.1	43.1
$C_{16}H_{18}$	10.0	38.9	84.1
$C_{18}H_{20}$	22.5	95.9	161.4

Iteration time in seconds for double precision.

Staticly distribute most diagrams between GPU and CPU, dynamically distribute leftovers. Small terms always done on CPU.

Details

1. Preallocate buffers on GPU, used pinned buffers on CPU.
2. Put GPU transfers and kernels on a stream.
3. Backfill with CPU computation.
4. See who finishes first, rebalance next iteration.

Complications:

- CUBLAS stream support took a while.
- Multi-GPU nodes and threads was a pain; now using 1 GPU per MPI rank.

Older gripes:

- Early CUBLAS required us to pad dimensions.
- GPUdirect two years later than I wanted it.

Lessons learned

- Do not GPU-ize legacy code!
  - Reimplementation from scratch was faster and easier.
  - Verification of new implementation is a challenge.
- Task-parallelism absolutely critical for heterogeneous utilization.
- Threading ameliorates memory capacity and BW bottlenecks. (How many cores required to saturate STREAM BW?)
- Data-parallel kernels very easy to implement in both OpenMP and CUDA.
- Careful organization of asynchronous data movement hides entire PCI transfer cost for non-trivial problems.
- Näive data movement leads to 2x; smart data movement leads to 8x.

Acknowledgments

Argonne NATIONAL LABORATORY

NERSC

Hardware Details

	CPU		GPU	
	X5550	2 X5550	C1060	C2050
processor speed (MHz)	2660	2660	1300	1150
memory bandwidth (GB/s)	32	64	102	144
memory speed (MHz)	1066	1066	800	1500
ECC available	yes	yes	no	yes
SP peak (GF)	85.1	170.2	933	1030
DP peak (GF)	42.6	83.2	78	515
power usage (W)	95	190	188	238

Note that power consumption is apples-to-oranges since CPU does not include DRAM, whereas GPU does.